Objektorientierte Programmierung

Kapitel 13: Exceptions

Stefan Brass

Martin-Luther-Universität Halle-Wittenberg

Wintersemester 2012/13

http://www.informatik.uni-halle.de/~brass/oop12/

Inhalt

Motivation (1)

- Methoden können manchmal mit Eingabewerten oder in Situationen aufgerufen werden, bei denen sie ihre Aufgabe nicht erfüllen können. Beispiele:
  - `Integer.parseInt("abc")`

    Die Eingabe muss eine gültige Repräsentation einer Zahl sein, also im wesentlichen eine Ziffernfolge. Der Benutzer kann aber beliebige Zeichenfolgen eingeben. Die Prüfung der Eingabe auf Korrektheit erfolgt am einfachsten bei der Umwandlung, sonst macht man doppelte Arbeit.
  - `Datum(32,-1,2013)`
  - Versuch, von einem leeren Stack einen Wert herunter zu nehmen (Operation `pop()`).
  - Versuch, in eine Datei zu schreiben, wenn die Platte voll ist.
  - Erzeugung eines neuen Objektes, wenn der Speicher voll ist.

Motivation (2)

- Typische (früher übliche) Behandlungsmöglichkeiten:
  - Es wird ein Fehlerwert zurückgeliefert, z.B. `-1`.
  - Das Programm wird mit einer Fehlermeldung beendet.
- Beide Möglichkeiten haben wesentliche Nachteile (siehe folgende Folien).
- Deswegen haben moderne Sprachen üblicherweise einen Exception-Mechanismus.

  Auch dies ist allerdings keine perfekte Lösung (s.u.).
- Die wesentliche Idee ist dabei, die Fehlererkennung und die Fehlerbehandlung zu trennen.

Motivation (3)

Probleme mit Fehlerwert:

- Manchmal gibt es keinen unbenutzten Wert, den man zur Signalisierung von Fehlern benutzen kann.

  `Integer.parseInt()` kann jeden möglichen `int`-Wert liefern. In solchen Situationen müßte man ein Objekt mit zwei Komponenten liefern: Dem eigentlichen Rückgabewert und dem Status-Indikator (Fehler oder ok).

- Diese Art der Fehlerbehandlung ist mühsam, weil bei jedem Aufruf abgefragt werden muss, ob er erfolgreich war.

  Das Programm wird dadurch deutlich länger.

- Diese Abfrage kann leicht vergessen werden.
  Dann wird mit einem falschen Wert weitergerechnet.

Motivation (4)

Probleme mit Programm-Abbruch:

- Die Methode, die den Fehler feststellt, weiß nicht, ob der Aufrufer noch etwas Wichtiges zu tun hat.
- Z.B. bei einem Editor / Spiel: Noch einmal abspeichern.

  Am besten in eine temporäre Datei, weil die Daten im Puffer ja durch den Fehler beschädigt sein könnten, und man den letzten sicheren Stand nicht überschreiben will. Aber wenn der Benutzer lange mit dem Programm gearbeitet/gespielt hat, möchte er möglichst vermeiden, dass durch einen Programmierfehler alles umsonst war. Bei dem Spiel Nethack wurde festgestellt, dass der in solchen Fällen gesicherte Spielstand für das Debugging oft nützlich war.
- Bei einem Kernkraftwerk: Anlage in einen sicheren Betriebszustand herunterfahren.

Inhalt

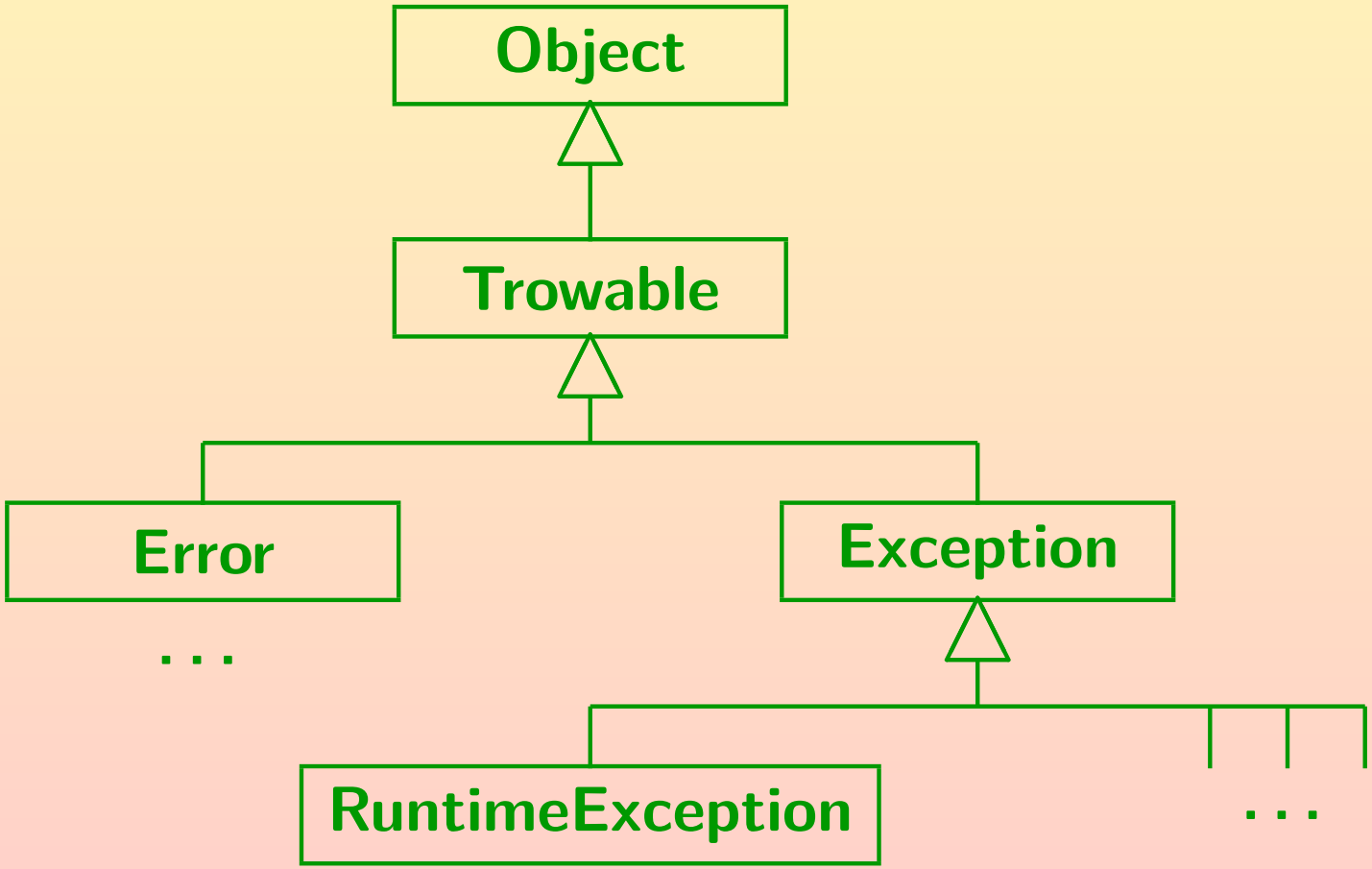
Exception-Klassen (1)

- In einem Programm können ganz unterschiedliche Fehler auftreten, die auch unterschiedlich behandelt werden müssen.
- Oft müssen dem Aufrufer auch zusätzliche Daten zum Fehler übermittelt werden.
- In Java werden für Exceptions (Fehler, Ausnahme-Situationen) daher Objekte erzeugt.

  So kann der bereits in der Sprache vorhandene Subklassen-Mechanismus genutzt werden, um auch Exceptions zu klassifizieren. Man kann auch eigene Exception-Klassen definieren.
- Diese Exception-Objekte sind in die Klassen-Hierarchie unterhalb der Klasse "`Throwable`" eingebettet.

  Exceptions werden mit dem Schlüsselwort "`throw`" ausgelöst. Mit dem Schlüsselwort "`catch`" kann man eine Fehlerbehandlung angeben (s.u.).

Exception-Klassen (2)



Exception-Klassen (3)

- Z.B. löst `Integer.parseInt()` im Fehlerfall eine `NumberFormatException` aus, das ist eine Subklasse von `IllegalArgumentException`, das wieder von `RuntimeException`.
- Wenn man auf ein Array außerhalb der Grenzen zugreift, erhält man eine `ArrayIndexOutOfBoundsException`, das ist Subklasse von `IndexOutOfBoundsException`, und wieder von `RuntimeException`.
- Wenn man mit `new FileInputStream(String name)` versucht, eine Datei zum Lesen zu öffnen, und es gibt die Datei nicht, bekommt man eine `FileNotFoundException`, Subklasse von `IOException`, und das wieder von `Exception`.

`IOException` und `FileNotFoundException` gehören zum Paket "`java.io`".

Exception-Klassen (4)

- Der Unterschied zwischen `Error` und `Exception` ist:
  - `Error` sind so schwere Fehler, dass sie in einem normalen Programm nicht behandelt werden können.

    Z.B. `VirtualMachineError`, `AssertionError`. Es ist nicht verboten, dafür eine `catch`-Klausel zu schreiben, aber es ist allgemein empfohlen, das nicht zu tun. Wenn die virtuelle Maschine nicht mehr richtig arbeitet, ist auch unklar, ob sie den `catch`-Block ausführen könnte.
  - `Exceptions` sind Ausnahmesituationen, für die sich ein normales Programm interessieren kann.

Exception-Klassen (5)

- Es gibt noch eine weitere Unterscheidung:
  - Subklassen von `Error` und `RuntimeException`, die eine Methode erzeugen könnte, müssen im Methodenkopf nicht deklariert werden (siehe Folie 38).

    Die Idee ist, das solche Fehler im Prinzip überall auftreten können (es handelt sich ja um Programmierfehler). Z.B. könnte jeder Array-Zugriff eine `ArrayIndexOutOfBounds`-Exception erzeugen. Es ist aber nicht verboten, solche Exceptions auch zu deklarieren. Z.B. weist die Dokumentation zu `String.parseInt()` darauf hin, dass eine `NumberFormatException` erzeugt werden könnte, obwohl dies eine indirekte Subklasse von `RuntimeException` ist.
  - Die anderen Exceptions ("checked exceptions") müssen dagegen im Methodenkopf deklariert werden.

    Sie sind spezifischer für bestimmte Fehlersituationen, und man kann erwarten, dass der Aufrufer sie behandeln will.

Exception-Klassen (6)

- `Throwable` hat eine Reihe nützlicher Methoden, z.B.:

  Siehe [http://docs.oracle.com/javase/6/docs/api/java/lang/Throwable.html]

  - `String getMessage()`
    Liefert einen Fehlermeldungstext ("detail message").
  - `String toString()`
    Name der Exeption-Klasse, ":", "detail message".
  - `void printStackTrace()`
    Zeigt Schachtelung der Methoden, deren Aufrufe zum Auslösen der Exception geführt haben.
  - `Throwable getCause()`
    Liefert die Exception, die zu dieser Exception geführt hat.

    Exceptions werden manchmal verkettet, so dass ein spezieller Fehler für eine bestimmte Operation nach oben in einer größeren und anwendungs-näheren Kategorie weitergegeben wird.

Exceptions, die man kennen sollte (1)

- Das folgende Beispiel-Programm demonstriert einige Exceptions (Laufzeit-Fehler), die bereits in früheren Kapiteln genannt wurden.

  So, wie ein erfahrener Programmierer die häufigsten Fehlermeldungen des Compilers kennt, sollte auch bei typischen Exceptions sofort klar sein, was sie bedeuten (die eigentliche Ursache zu finden, kann dann etwas dauern).

- Im Beispiel-Programm wird für jede Methode im Kopf deklariert, welche Exception sie erzeugen kann.

  Die Methoden sind so gemacht, dass sie die Exception auch wirklich erzeugen.

- Die Ausgabe-Anweisung wird jeweils nicht ausgedruckt, weil die Ausführung der Methode aufgrund der Exception abgebrochen wird.

  Es findet ein nicht-lokaler Sprung zum `catch`-Block im Hauptprogramm statt.

Exceptions, die man kennen sollte (2)

```
(1)  // Fuer einige Beispiele ist eine Klasse
(2)  // mit Subklasse noetig:
(3)  class Ober { int a = 1; }
(4)  class Unter extends Ober { }
(5)
(6)  class ExceptionTest {
(7)
(8)      static void test1()
(9)          throws ArrayIndexOutOfBoundsException
(10)     {
(11)         int[] a = new int[5];
(12)         int i = 5;
(13)         int n = a[i]; // Exception
(14)         System.out.println("Nie ausgeführt.");
(15)     }
(16)
```

Exceptions, die man kennen sollte (3)

```
(17)    static void test2()
(18)        throws StringIndexOutOfBoundsException
(19)    {
(20)        String s = "abc";
(21)        int i = 3;
(22)        char c = s.charAt(i); // Exception
(23)        System.out.println("Nie ausgeführt.");
(24)    }
(25)
(26)    static void test3()
(27)        throws NullPointerException
(28)    {
(29)        Ober o = null;
(30)        o.a = 2; // Exception
(31)        System.out.println("Nie ausgeführt.");
(32)    }
(33)
```

Exceptions, die man kennen sollte (4)

```
(34)    static void test4()
(35)        throws ClassCastException
(36)    {
(37)        Ober o = new Ober();
(38)        Unter u = (Unter) o; // Exception
(39)        System.out.println("Nie ausgeführt.");
(40)    }
(41)
(42)    static void test5()
(43)        throws ArrayStoreException
(44)    {
(45)        Ober o[] = new Unter[5];
(46)        o[0] = new Ober(); // Exception
(47)        System.out.println("Nie ausgeführt.");
(48)    }
(49)
```

Exceptions, die man kennen sollte (5)

```
(50)      static void test6()
(51)          throws ArithmeticException
(52)      {
(53)          int i = 0;
(54)          int n = 5 / i; // Exception
(55)          // Bei double keine Exception (-> NaN)
(56)          System.out.println("Nie ausgeführt.");
(57)      }
(58)
(59)      static void test7()
(60)          throws NumberFormatException
(61)      {
(62)          String s = "abc";
(63)          int n = Integer.parseInt(s); // Excep.
(64)          System.out.println("Nie ausgeführt.");
(65)      }
(66)
```

Exceptions, die man kennen sollte (6)

```
(67)      public static void main(String[] args) {
(68)         for(int i = 1; i <= 7; i++) {
(69)            System.out.print("test"+i+"(): ");
(70)            try {
(71)               switch(i) {
(72)                  case 1: test1();
(73)                  case 2: test2();
...                   ...
(79)               }
(80)            }
(81)            catch(Exception e) {
(82)               System.out.println(e);
(83)            }
(84)         }
(85)      }
(86)
(87)   }
```

Inhalt

try-catch (1)

- Wenn man damit rechnet, dass eine Exception in einer Anweisung auftreten kann, und man darauf reagieren will, schreibt man sie in einen "try-catch"-Block:

```
try {
    int n = Integer.parseInt(args[0]);
    verarbeite(n);
} catch(NumberFormatException e) {
    System.out.println("Eingabeformat falsch");
}
```

- Falls bei der Ausführung einer Anweisung eine Exception auftritt, wird sie sofort abgebrochen, und es findet ein Sprung zu einem weiter außen liegenden `catch`-Block statt (mit passender Exception-Klasse).

  Dieses Verhalten ähnelt einer `break`-Anweisung.

try-catch (2)

- Im Beispiel wird der Aufruf "`verarbeite(n)`" also nicht ausgeführt, wenn bei der Umwandlung des Argumentes ein Fehler auftrat.
- Der Sprung zu einem Exception-Handler (`catch`-Block) findet auch über Methodengrenzen hinweg statt:
  Wenn man die `try-catch`-Anweisung nicht verwendet,
  - wird die Ausführung der Methode abgebrochen,
  - und es wird in der aufrufenden Methode nach einem passenden Exception Handler gesucht.
  - Gibt es auch dort nichts, werden weiter Methoden verlassen.
  - Hat schließlich auch `main` keinen Exception Handler, wird das Programm mit einer Fehlermeldung beendet.

try-catch (3)

```
(1)  class Sprung {
(2)
(3)      static int g() {
(4)          System.out.println("In g() ...");
(5)          int i = 0;
(6)          int n = 1 / i; // Hier passiert's
(7)          System.out.println("Nicht gedruckt.");
(8)          return 1; // Nicht ausgeführt
(9)      }
(10)
(11)     static int f() {
(12)         System.out.println("In f() ...");
(13)         int j = g();
(14)         System.out.println("Nicht gedruckt");
(15)         return 2; // Auch nicht ausgeführt
(16)     }
(17)
```

try-catch (4)

```
(18)      public static void main(String[] args) {
(19)          System.out.println("Anfang.");
(20)          try {
(21)              int k = f();
(22)              System.out.println("Nicht gedruckt");
(23)          }
(24)          catch(Exception e) {
(25)              System.out.println("Exception!");
(26)              e.printStackTrace();
(27)          }
(28)          System.out.println("Ende.");
(29)      }
(30)  }
```

try-catch (5)

- Das Programm erzeugt folgende Ausgaben:

```
Anfang.
In f() ...
In g() ...
Exception!
java.lang.ArithmeticException: / by zero
        at Sprung.g(Sprung.java:6)
        at Sprung.f(Sprung.java:13)
        at Sprung.main(Sprung.java:21)
Ende.
```

- Weil `f()` und `g()` jeweils keinen Exception Handler haben, springt die Ausführung von Zeile 6 zu Zeile 24 und verlässt dabei zwei Methodenaufrufe auf nicht-normalem Wege.

  Deswegen spricht man auch von einem “nichtlokaler Sprung”.

try-catch (6)

Syntaktische Hinweise:

- Nach `try`, `catch` und `finally` muss jeweils ein Block stehen.

  Auch wenn es sich nur um ein einzelnes Statement handelt, muss man die Klammern `{...}` schreiben.

- In den Klammern vom "`catch(...)`" muss man einen Parameter wie `e` deklarieren, selbst wenn man ihn nicht verwendet.

  Es reicht nicht, nur die Klasse anzugeben.

- Man darf in einem `catch`-Block auch eine neue Exception auslösen.

  Oder auch die gleiche Exception weitergeben. Siehe `throw`-Anweisung unten.

try-catch (7)

- Die Exception Handler werden der Reihe nach durchgegangen, und es wird der erste ausgeführt, bei dem die angegebene Klasse eine Oberklasse der tatsächlich aufgetretenen Exception ist.

  Man kann z.B. vorne spezielle Exception-Klassen angeben, und zum Schluss noch eine allgemeine Klasse wie `Exception`, um "alle anderen" Exceptions zu behandeln.

- Nach Ende der Ausführung des Exception Handlers (also des `catch`-Blockes) wird mit der Ausführung des Programmcodes hinter der `try-catch`-Anweisung fortgefahren.

  Bzw. zuerst der `finally`-Block, falls vorhanden (s.u.). Es werden nicht eventuell weitere `catch`-Blöcke der gleichen `try-catch`-Anweisung betreten, auch wenn deren Klasse auch eine Oberklasse der aufgetretenen Exception ist.

Beispiel (1)

- Das folgende Beispiel zeigt das Einlesen einer Zahl von der Standard-Eingabe (Console).

  `System.in` hat den Typ `InputStream` und erlaubt nur, Bytes einzulesen. `InputStreamReader` wandelt es in Zeichen (`char`) um. `BufferedReader` liest eine größere Menge von Zeichen auf einmal und speichert sie zwischen.

```
(1)  import java.io.*;
(2)
(3)  class Quadrat {
(4)
(5)      // Beispiel fuer Verwendung der Eingabe:
(6)      // Quadratzahl ausdrucken.
(7)      static void verarbeite(int n) {
(8)          int q = n * n;
(9)          System.out.println(n + "^2 = " + q);
(10)     }
(11)
```

Beispiel (2)

```
(12)     public static void main(String[] args) {
(13)         BufferedReader inBuf =
(14)             new BufferedReader(
(15)                 new InputStreamReader(System.in));
(16)         try {
(17)             String eingabe = inBuf.readLine();
(18)             int n = Integer.parseInt(eingabe);
(19)             verarbeite(n);
(20)         } catch(NumberFormatException e) {
(21)             System.out.println(
(22)                 "Eingabeformat falsch");
(23)         } catch(IOException e) {
(24)             System.out.println(
(25)                 "Fehler beim Einlesen");
(26)         }
(27)     }
(28) }
```

finally (1)

- Nach den catch-Klauseln kann man noch einen finally-Block angeben:

```
try {
    ...
} catch(...) {
    ...
} catch(...) {
    ...
} finally {
    ...
}
```

- Es ist auch möglich, finally ohne catch zu verwenden.

`finally` (2)

- Die Anweisungen im `finally`-Block werden auf jeden Fall ausgeführt,
  - sowohl, wenn eine Exception bei der Ausführung des `try`-Blockes aufgetreten ist,

    In diesem Fall nach der Ausführung des passenden Exception Handlers (falls die Klasse in einer der `catch`-Klauseln passte, sonst direkt nach dem `try`-Block).
  - als auch, wenn die Ausführung des `try`-Blocks normal bis zum Ende gekommen ist,
  - sogar, wenn der `try`-Block mit `break` oder `return` verlassen wurde.

finally (3)

- Der Zweck des `finally`-Blocks ist es, eventuelle Abschluss-Arbeiten noch durchführen zu können, bzw. belegte Ressourcen wieder frei zu geben.
- Wenn man z.B. eine Datei eröffnet hat, sollte man die wieder schließen.

  Die Gesamtanzahl der gleichzeitig offenen Dateien pro Prozess ist durch das Betriebssystem begrenzt (z.B. auf 15). Wenn man immer wieder Dateien öffnet, ohne sie zu schließen, kommt man in Schwierigkeiten.
- Es wäre mühsam und fehleranfällig, jeden möglichen Ausführungspfad zu bedenken.

  Deswegen ist es so nützlich, dass `finally` auf jeden Fall ausgeführt wird.

finally (4)

- Beim Betreten des `finally`-Blockes wird gemerkt, welches Ende die bisherige Ausführung hatte, z.B., dass es ein `return` mit einem bestimmten Wert war.
- Wenn der `finally`-Block normal endet, wird dem gemerkten Ende fortgefahren.

  Das `return` findet also noch statt, der `finally`-Block schiebt sich nur dazwischen.
- Der `finally`-Block kann aber auch selbst ein `return` durchführen oder eine Exception auslösen, dann wird das gemerkte Ende überschrieben.

Inhalt

Auslösen von Exceptions mit `throw` (1)

- Man löst eine Exception aus mit der Anweisung

  `throw` ⟨Expression⟩;

- Dabei muss die Expression ein Objekt der Klasse `Throwable` oder einer ihrer Unterklassen liefern.

- Beispiel:

```
String meldung = "Falsche Kanal-Nummer: " + kanal;
throw new IndexOutOfBoundsException(meldung);
```

- Wie `break` oder `return` unterbricht `throw` den normalen Fluss der Ausführung, direkt folgende Statements würden niemals ausgeführt.

  Ein `throw` könnte also z.B. am Ende eines `if`/`else`-Blockes stehen.

  Ein `throw` am Ende des Methoden-Rumpfes ist auch möglich. Man braucht dann kein `return`, weil die Methode nicht normal verlassen wird.

Auslösen von Exceptions mit `throw` (2)

- `Throwable` hat vier Konstruktoren, davon kann man die ersten zwei für die meisten Subklassen erwarten:

  Die anderen beiden nur, wenn Verkettung von Exceptions Sinn macht.

  Ausnahme z.B.: `java.text.ParseException(String s, int offset)` einziger Konstruktor dieser Klasse.

  - `Throwable()`
  - `Throwable(String message)`
  - `Throwable(String message, Throwable cause)`
  - `Throwable(Throwable cause)`

- Man kann auch eine mit `catch` "aufgefangene" Exception erneut auslösen, um sie an die aufrufende Methode weiterzugeben:

```
throw e;
```

Deklaration von Exceptions im Methoden-Kopf (1)

- Der Aufrufer einer Methode muss eventuell berücksichtigen, dass die Methode auch nicht-normal enden kann.
- Ein normaler Methoden-Aufruf kann dann auch dazu führen, dass die eigene Methode unerwartet verlassen wird.
- Wenn man z.B. eine Datenstruktur ändert, und temporär einen ungültigen Zwischenzustand hat, ist es wichtig, dass die Ausführung nicht einfach abgebrochen wird, bevor man wieder einen gültigen Zustand hergestellt hat.
- Deswegen sollte im Methoden-Kopf deklariert werden, welche Exceptions die Methode erzeugen kann.
- Für `Error` und `RuntimeException` und ihre Subklassen ist dies optional, für alle anderen Exceptions Pflicht.

Deklaration von Exceptions im Methoden-Kopf (2)

- Beispiel:

```
void startZeit(int kanal)
    throws IndexOutOfBoundsException {
    ...
}
```

- Diese Exception müßte nicht unbedingt deklariert werden, da sie eine Subklasse von `RuntimeException` ist.
- Man kann auch mehrere Exception-Klassen angeben, durch Komma getrennt.

  Die "`throws`"-Klausel sagt nur aus, dass die Methode möglicherweise eine Exception dieses Typs erzeugen könnte. Man bekommt keine Fehlermeldung, wenn dieser Fall tatsächlich nie eintreten kann. Es wäre aber schlechter Stil, unmögliche Exceptions zu deklarieren. Man erzeugt damit auch eine zusätzliche Last für den Aufrufer (siehe nächste Folie).

Deklaration von Exceptions im Methoden-Kopf (3)

- Wenn man eine Methode `m()` aufruft, die eine Exception vom Typ `E` erzeugen kann, muss man
  - die Exception selbst behandeln, d.h. `m()` in einem `try`-Block aufrufen, zu dem ein Exception-Handler für `E` (oder eine Oberklasse von `E`) gehört, oder
  - deklarieren, dass die eigene Methode Exceptions vom Typ `E` (oder einer Oberklasse von `E`) erzeugen kann.
- Dies ist die sogenannte "catch-or-throw" Regel.
- Vergisst man es, erhält man eine Fehlermeldung der Art:

```
ExDeclErr.java:4:
        unreported exception java.io.IOException;
        must be caught or declared to be thrown
```